Optimization of Fusion Based Image Upscaling For Video Games

Kyler Martinez
Electrical and Computer Engineering
University of Arizona
Tucson, AZ
kylerm23@arizona.edu

Matthew Toro
Electrical and Computer Engineering
University of Arizona
Tucson, AZ
matthewt1441@arizona.edu

***Abstract*—With video games increasing in popularity in modern society, techniques to improve performance while maintaining quality are highly sought after. In this paper, a modified version of the image fusion upscaling method proposed by X. Kang, S. Li, and J. Hu [3] was implemented as an upscaling technique for video game applications. GPU optimization techniques such as shared memory tiling, separable convolution, memory coalescing, kernel merging, and pipelining were used to drastically improve the speed and throughput of the fusion algorithm. Experimental results show that, when compared to a naive CUDA implementation, the optimized algorithm performs up to 12x faster. This may allow this version to be implemented into the rendering pipeline with minimal additional overhead. While performance was greatly increased overall visual quality of the image was not, due to the upscaling methods used for the fusion algorithm.**

***Index Terms*—Image Super Resolution, CUDA, Image up scaling, BiCubic Interpolation, Nearest Neighbors, Fusion Upscaling, Data streaming**

I. Introduction

Video games have pushed processing power to extremes with more advanced and realistic physics and increasingly high resolutions and object quality. This has caused a great deal of difficulty in achieving higher resolutions and maintaining high frame rates (FPS). One technique used to improve frame rate with comparable image quality is to render at a lower resolution and upscale it to the desired resolution. Rendering the game at a lower quality and upscaling the image using advanced techniques such as NVIDIA's DLSS have proven to be highly effective. Despite the improved frame rates, the upscaled images contain artifacts not found in native rendering. However, these artifacts may be relatively minor depending on the preference of the user and the game.

One of the greatest challenges for upscaling methods is optimizing the upscaling algorithm because the overall frame rate is a combination of the native rendering (N) and the upscaling algorithm (U) (1).

$$\text{Effective FPS} = \frac{U * N}{U + N} \tag{1}$$

For example, if the game renders at 120 FPS, and the upscaling algorithm renders at 120 FPS, the overall frame rate is reduced to 60 FPS. The upscaling algorithm will bottleneck the performance of the system and prevent the game from running as fast as possible. In the worst case, native rendering may outperform the performance of the entire system. Due to this, optimization of the upscaling algorithm is imperative.

In this paper, we will implement a modified version of an upscaling method proposed by X. Kang, S. Li, and J. Hu [3], a fusion algorithm. This method utilizes two image interpolation techniques and fuses their generated frames to negate the artifacts produced by both methods.

II. Background

A. Image Upscaling

Upscaling methods vary based on the complexity, and some examples include single-image super-resolution (SISR) and temporal upscaling. SISR upscaling creates the new image using information found in the image whereas temporal upscaling methods utilize information from previous frames to create new ones. For this study we focused on SISR upscaling methods to avoid ghosting and other side effects found in temporal methods. Regardless of the upscaling method, the goal for the upscaling method is to utilize a less computationally expensive algorithm relative to native high resolution rendering at the trade off of introducing artifacts into the images.

The simplest upscaling algorithm is Nearest Neighbors which copies the "nearest" input pixel into the output image. Fig. 1 showcases an example of Nearest Neighbors upscaling where the single pixel corner generates the data for four pixels in the output.

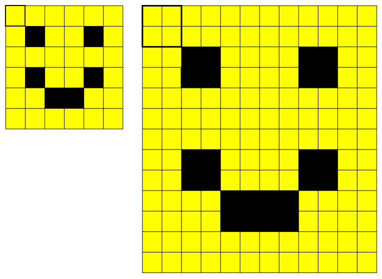

Fig. 1. Nearest Neighbors Upscale 2x.

The input image coordinates corresponding to the output image can be found with the following equation:

$$\begin{aligned} Input\ X &= (Output\ X)//Scale \\ Input\ Y &= (Output\ Y)//Scale \end{aligned} \tag{2}$$

The Scale refers to the upscaling factor for the image and the // operator is integer division, it will disregard the decimal result from the division operation.

BiCubic interpolation is another technique used to upscale images. It relies on the intensity values of neighboring pixels to generate a set of cubic functions that are used to interpolate new output pixels. This method results in a smoother output image, but can have its own set of artifacts. Fig. 2 shows how the BiCubic Interpolation technique uses the nearest set of 4x4 pixels to generate a set of output pixels, shown in blue. When at the edge of an input image, this technique can utilize the nearest-neighbor input pixels to fill the window needed to generate the cubic functions.

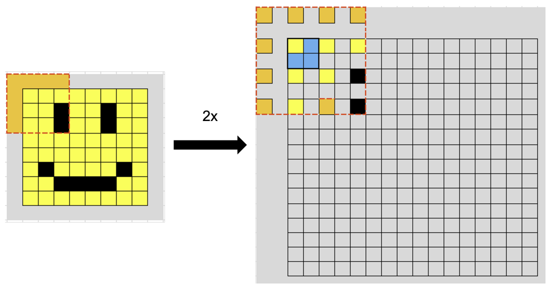

Fig. 2. BiCubic Upscale 2x.

Cubic functions are first generated across the x direction and the x values of the output pixels are passed into these for functions. This is then used to generate a Cubic function in the y direction, and the y values of the output pixels are passed into this function to generate the final output pixel value. Fig. 3 provides a visualization of how this technique works.

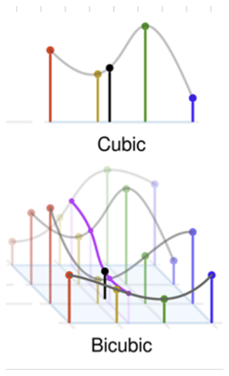

Fig. 3. BiCubic Interpolation [14]

B. Fusion Algorithm

Image fusion can be used to improve the overall quality of upscaling. First a low resolution (LR) image is upscaled using two different algorithms. The artifacts within two upscaled images can be detected and removed, and these images can then be fused together to take the best features from each. An artifact map can help determine which pixels from each upscaled image should be fused together into one. To generate this artifact map, first a grayscale image of both upscaled images is created. These images are then used to generate a Structural Similarity Index Measure (SSIM) map, which is combined with a difference map to highlight the major artifact differences between two upscaled images. This map is then run through a blurring filter to reduce noise, and a threshold is applied to binarize the artifact map for image fusion. Fig. 4 shows the flow chart of the fusion algorithm we implemented.

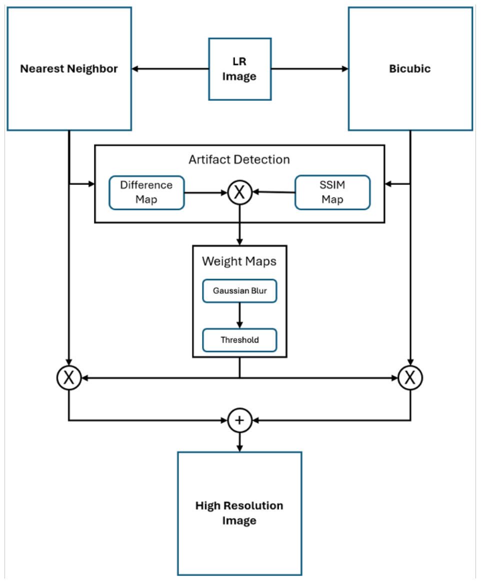

Fig. 4. Fusion Algorithm

The Structural Similarity Index Measure will return the similarity between two images and can be calculated after computing the mean of each image and variance of each image. Wang and his team utilize the following equation to compute the SSIM index for image x and image y [4]:

$$SSIM = \frac{(2\mu_x\mu_y + C_1)(2\sigma_{xy} + C_2)}{(\mu_x^2 + \mu_y^2 + C_1)(\sigma_x^2 + \sigma_y^2 + C_2)} \qquad (3)$$
$$C_1 = 6.5025,\ C_2 = 58.5225$$

The artifact map can be generated by using the SSIM index on a small window (8x8) and the absolute difference of the two images. In the following equation, we calculate the artifact map's value for each pixel to be the difference between the images multiplied by the SSIM index for the small window.

$$Artifact[x][y] = SSIM * (|IMG_1[x][y] - IMG_2[x][y]|) \quad (4)$$

A 7x7 Gaussian Blur filter is applied to the artifact map and a threshold of 0.05 was used to binarize the map as it produces the cleanest map with the least amount of noise. Finally, the artifact map is then used to combine both upscaled images using the following equation:

$$IMG_{fused} = IMG_1 * Artifact + IMG_2 * (1 - Artifact) \quad (5)$$

C. Programming with CUDA

Image upscaling in general is suitable for parallelization on the GPU. Generally, the same operation will occur on every individual pixel or the same operations will be applied to the same group of pixels.

CUDA is an extension of the C++ programming language that allows programmers to control and utilize GPU resources. Each of the steps of the Image Fusion algorithm is divided into kernels that are executed on a GPU.

For our evaluation, we utilized the Tesla P100 GPU to time the algorithms and optimizations created. The primary metric used to evaluate the performance of the algorithms is the execution time or FPS for the kernel. In addition to the frame rate, we utilized NVIDIA Visual Profiler (NVVP) to analyze execution time along with GPU utilization. Lastly, we estimated the maximum FLOPs or bandwidth for the kernels developed to determine if they were **Compute Bound** or **Memory Bound**, meaning they are limited by the FLOPs of the GPU or its bandwidth. The Tesla P100 used has a memory bandwidth of 732 GB/s and a maximum FLOPs of 9.526 TFLOPs. The realized FLOPs and bandwidth are a function of the number of FLOP in the kernel and the maximum number of read or write operations performed. For our testing, each memory transaction requires 4 bytes of data. The equation below describes the calculation for the FLOPs and bandwidth for each kernel; whichever calculation produces a value under the theoretical limit determines whether it is compute or memory bound (6) (7).

$$\text{FLOPs} = 720\ \text{GB/s} * \frac{\text{FLOP}}{\text{MemoryAccess} * 4\text{B}} \tag{6}$$

$$\text{Bandwidth} = 9.526\ \text{TFLOPs} * \frac{\text{MemoryAccess} * 4\text{B}}{\text{FLOP}} \tag{7}$$

Generally, our optimizations fall into the following categories:

- Memory Coalescing
- Kernel Merging
- Batching
- FLOP Reduction
- Shared Memory
- Separable Convolution

D. Related Work

No studies were found that implemented this fusion algorithm targeting GPUs. Due to this, no baseline results were available to compare our solutions to. However, some of the individual algorithms used within the fusion technique have been analyzed and shown to have significant performance improvements when implemented in CUDA.

Current algorithmic image interpolation techniques are limited based on whether they are compute intensive or data intensive interpolations. Situations with higher upscale factors will have more compute intensive interpolations because less image data will be used to calculate significantly more output pixels. In the case of video game rendering, this situation will occur with smaller rendering resolutions being upscaled to high resolutions such as 4K. Data intensive interpolations will occur with smaller upscaling factors as less computation is required to generate more pixels and the limiting factor shifts towards data access. The computing and data bottlenecks will be exasperated by the scaling factor and input resolution respectively. Implementations of various upscaling algorithms have been created for serial execution on CPUs as well as other computing platforms, such as FPGAs and ASICS. GPU Implementations of Nearest Neighbor and BiCubic Interpolation have shown improvements of x1.92 and x8.49 compared to serial implementation in data intensive interpolation and 5.70 and 17.44 for compute intensive interpolations [6].

The SSIM is a more computationally intensive portion of the artifact detection algorithm and can be improved with the use of tiling and shared memory. Using common optimization techniques such as reduction and shared memory, SSIM has been shown to have a speedup of x10 when compared to a serial calculation [9].

The Gaussian Blur filter is an important step that reduces noise generated by the SSIM and difference maps, which in turn produces a cleaner weight map for image fusion. CUDA optimization techniques such as shared/constant memory utilization, tiling, and separable convolution have been shown to produce speed ups of 200x when compared to a C implementation [11].

III. Baseline Solutions

A. Serial Solution

A serial implementation of the fusion algorithm performs incredibly poorly, producing frame rates below 1 FPS. Table I contains the FPS for the serial code averaged across 10 frames. The serial solution is unusable in any realistic application and would require multiple seconds to produce an output frame.

B. Naive CUDA Solution

A Naive implementation of the fusion algorithm was created by simply replacing most for loops of the serial implementation with the threads of a GPU. Returning to Table I, the naive solution produced a frame rate 100 to 250 times greater than the serial solution. This version drastically improved the performance of the fusion algorithm when compared to the serial code, but it still does not provide enough performance to be utilized in any realistic application.

TABLE I
Baseline Frame rate

Input Image	**Scale Factor**	**Baseline FPS**	
		Serial Code	**Naive CUDA**
640x480	**2**	**0.893**	**89.6**
	4	**0.223**	**30.9**
	6	**0.098**	**14.5**
1280x720	**2**	**0.297**	**35.1**
	4	**0.074**	**9.9**
	6	**0.033**	**4.5**
1920x1080	**2**	**0.132**	**16.2**
	4	**0.033**	**4.3**
	6	**0.015**	**2.0**

IV. Project Development

A. Milestones

Throughout the development of the algorithm we created generations of the fusion algorithm:

- Serial Solution
  - C++ solution executed only on a CPU
- Naive Solution
  - CUDA C translation of serial code into kernels. No optimizations in computations.
- Basic Optimizations
  - Incorporates optimizations such as memory coalescing and kernel merging.
- Advanced Optimizations
  - Includes optimizations such as shared memory, separable convolution, and FLOP reductions.
- Proposed Solution
  - Utilizes the best performing kernels for each step of the algorithm.

B. RGBA Data Structure

To improve memory accesses the RGBA structure was created to read one 32-bit value per pixel as opposed to reading three separate Red (R), Green (G), and Blue (B) pixel values. The data structure allows for an alpha channel (A) but this was disregarded. Fig. 5 describes the RGBA data type and how the data is stored in memory.

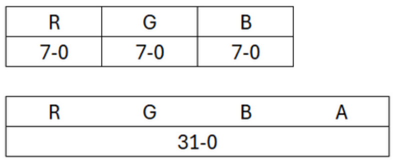

Fig. 5. RGBA Data Structure.

The RGBA structure helps simplify the data mapping for each thread as each thread can read in the entire pixel value at once rather than each channel individually. This also allows for greater memory access utilization as the RGBA struct is 32 bits long, providing the alignment in global memory needed for coalesced memory access.

V. Nearest Neighbors Optimization

A. Nearest Neighbors Kernels

Nearest Neighbors is a relatively simple algorithm and we were able to apply the following strategies to optimize the kernel:

- Memory Coalescing
- Kernel Merging
- Shared Memory

Using these optimization strategies we created three solutions solutions utilizing these strategies:

- Basic RGBA Solution
  - Uses the RGBA struct to improve memory coalescing and kernel merging to calculate the color and grayscale image at the same time to reduce global memory reads.
- Shared Memory: One Thread Per Output (OTPO)
  - One thread is assigned to each output pixel. Only the threads needed will access the shared memory.
- Shared Memory: One Thread Per Input (OTPI)
  - One thread is assigned per input image and writes to multiple output pixels. All threads access the shared memory.

B. Nearest Neighbors Results

For each kernel, we determined the ideal square block configuration and experimented with 8x8, 16x16, and 32x32 block configurations to ensure that it was a multiple of the warp size. Regardless of the scale factor, the results of the block dimension were the same and Table II showcased the execution time for each optimized kernel with different block dimensions. 16x16 proved to perform the best across the scaling factors and input sizes and was the configuration we proceeded with for kernel comparison.

TABLE II
NEAREST NEIGHBORS EXECUTION TIME: SCALE FACTOR 2, INPUT IMAGE 1920X1080

Kernel Type	Block Dim	Time [ms]
Basic	8x8	0.307
	16x16	0.253
	32x32	0.258
OTPO	8x8	0.299
	16x16	0.154
	32x32	0.167
OTPI	8x8	0.375
	16x16	0.379
	32x32	0.409

Table III contains the execution time data for every kernel using the 16x16 block dimension, 4x upscaling factor, and each image size. From these data, the Basic kernel performs the best for each image size. However, when the upscaling factor is 2x (Table II), the OTPO solution performs the best.

TABLE III
NEAREST NEIGHBORS EXECUTION TIME: SCALE FACTOR 4, BLOCK DIMENSION 16X16

Kernel Type	Input Size	Output Size	Time [ms]
Serial	640x480	2560x1920	127.328
	1280x720	5120x2880	593.070
	1920x1080	7680x4320	1107.524
Basic	640x480	2560x1920	0.152
	1280x720	5120x2880	0.427
	1920x1080	7680x4320	0.942
OTPO	640x480	2560x1920	0.175
	1280x720	5120x2880	0.488
	1920x1080	7680x4320	1.061
OTPI	640x480	2560x1920	0.213
	1280x720	5120x2880	0.612
	1920x1080	7680x4320	1.347

C. Nearest Neighbors Analysis

The basic solution on average performed 800x that of the serial execution time and outperformed the other kernels for all scale factors aside from 2x. The basic kernel outperforms the Naive kernel due to utilizing memory coalescing with the RGBA struct. There is little thread divergence except for the edges of the image, but there are redundant memory accesses from the global memory.

The OTPO shared memory solution suffers from the size of its shared memory shrinking as the scale size decreases and fewer threads participate in accessing the shared data. The shared memory is prone to bank conflicts as the data reuse increases and more threads will access the same address at the same time. The kernel performs the best when upscaling by a factor of 2x which is the configuration that ensures that the most threads are writing to the global memory. The OTPI shared memory solution utilizes all threads but results in more global memory writes as each thread will need to perform Scale^2 number of writes. The multiple write phases will decreases the amount of work that can execute in parallel and increase the time the kernel takes to process.

Regardless of the kernel used, the algorithm is memory bound due to the few floating point operations occurring in the kernel. 5 FLOP operations are required to compute the gray scale pixel for the two global memory writes for the output images. As a result, the GPU will process at approximately 0.458 TFLOPs which is 20.8 times less than the theoretical max.

To address the performance, the kernel could be improved with a more complex data mapping skill to ensure that all threads can participate in the data accessing stage and resemble some shared memory tiling kernels that compute in multiple phases. However, the best solution may be to merge with another kernel, like BiCubic, which will access the same data removing a global memory access per pixel.

VI. BiCubic Interpolation Optimization

A. BiCubic Kernels

The following strategies were used to optimize the BiCubic interpolation algorithm:

- Memory Coalescing
- Kernel Merging
- Shared Memory

Using these strategies we implemented two versions of the algorithm:

- Basic RGBA Solution
  - Uses the RGBA struct to improve memory coalescing and kernel merging to calculate the color and grayscale image at the same time to reduce global memory reads.
- Shared Memory: One Thread Per Output
  - One thread is assigned to each output pixel. Each thread will be used to fill the share memory tile with input image pixels. This will require striding as more input data is needed than output threads available. Each thread generates its own 4x4 window to interpolate over. Fig 6 shows how data was stored into shared memory and utilized to generate the upscaled image.

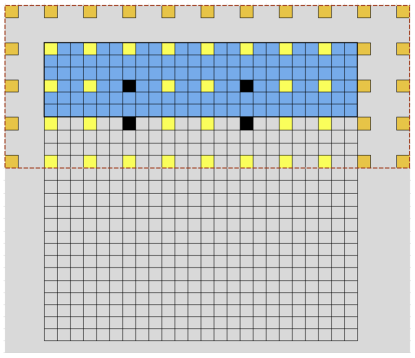

Fig. 6. BiCubic Shared Memory (Scale 3)

B. BiCubic Results

Because of the nature of the BiCubic algorithm the appropriate block dimensions should be a square that is a multiple of the scale factor. This ensures that each block has enough threads to fully utilize the 4x4 window of pixels used for the interpolation, as each window generates Scale x Scale number of output pixels. A block dimension of 4 x Scale allowed for the right balance of data reuse, while also ensuring enough blocks can run. Table IV shows the execution time for both versions of the algorithm using the 4 x Scale block dimensions and compared the results to the serial execution.

TABLE IV
BiCubic Execution Time

Kernel	Input Res	Scale	Time [ms]
Serial	**640x480**	**2**	**542.79**
Basic			**0.265**
Shared			**0.215**
Serial		**4**	**2175.63**
Basic			**1.024**
Shared			**0.924**

As expected, the shared memory kernel performed the best.

C. BiCubic Analysis

The Shared Memory version performed between 2000x and 2500x faster than the serial version of the BiCubic Algorithm. The utilization of shared memory is a huge performance boost as the same amount of input pixels are utilized to generate more output pixels. However, when analyzing the GPU utilization of this algorithm, its clear that this version of the algorithm is compute bound. 292 FLOPS are performed per thread to generate one output pixel, resulting in a maximum bandwidth of 261GB/s. This is about 2.8x less than the maximum bandwidth of the Tesla P100 GPU.

VII. Artifact Detection Optimization

A. Artifact Detection Kernels

The artifact detection kernel is compute heavy and utilized the following optimizations in our experiments:

- Kernel Merging
- Shared Memory
- Computation Reduction
- Data Batching

Using these optimization strategies we created three solutions solutions utilizing these strategies:

- Basic Solution
  - Merges the SSIM and difference map computation into one kernel. Uses a 16x16 block dimension.
- Shared Memory: All Threads Compute SSIM (ATCS)
  - All threads redundantly computes the SSIM for the window using a 8x8 block dimension.
- Shared Memory: One Thread Computes SSIM (OTCS)
  - One thread computes the SSIM for the window using a 8x8 block dimension.
- Shared Memory: Multiple Maps (MM)
  - The block reads four 8x8 windows into shared memory and computes all the windows at the same time using a 16x16 block dimension.

B. Artifact Detection Results

The artifact detection kernel's execution is independent of the scale factor and is dependent solely on the size of the input image generalized as total pixels. Table V contains the execution time data for every kernel using using three input image sizes. From our experiments, the ATCS solution proved to be the best kernel for computing the artifact map.

TABLE V
Artifact Detection Execution Time

Kernel Type	Pixels [MP]	Time [ms]
Serial	**1.23**	**82.295**
	8.29	**470.670**
	33.18	**1860.167**
Basic	**1.23**	**0.689**
	8.29	**4.484**
	33.18	**17.678**
ATCS	**1.23**	**0.351**
	8.29	**2.187**
	33.18	**8.720**
OTCS	**1.23**	**0.481**
	8.29	**3.149**
	33.18	**12.421**
MM	**1.23**	**0.432**
	8.29	**2.736**
	33.18	**10.825**

C. Artifact Detection Analysis

The ATCS solution on average performed 200x that of the serial execution time and 2x the Naive solution. Each of the kernels reduce the number of global memory accesses from 128 (64 for each image) to 2 (1 for each image). The ATCS solution excels at reducing the amount of memory accesses without synchronization steps but at the cost of redundant computation steps.

The OTCS solution reduced the amount of computations needed for every thread but due to the 8x8 window, only one warp was spared from calculating the SSIM. The additional synchronization and divergence prevented the OTCS solution from outperforming the ATCS solution.

The MM solution performed better than the OTCS solution but worse than the ATCS solution. There is evidence that a larger block dimension could improve the execution but the additional overhead to manage multiple maps hinders the kernels design.

The artifact detection kernels are computationally intensive and as a result are compute bound rather than memory bound. The SSIM calculation for ATCS, shown in Table VI, requires 540 FLOP per thread and two memory reads and one memory write. The kernel results in a maximum theoretical bandwidth of 141 GB/s, 5 times less than the maximum bandwidth provided by the P100 GPU.

TABLE VI
Execution Steps for SSIM

Step	Operations
MEM Read IMG1, IMG2	**2**
Image Difference	**0**
sum(IMG1)	**64**
sum(IMG2)	**64**
sum(IMG1 * IMG1)	**128**
sum(IMG2 * IMG2)	**128**
sum(IMG1 * IMG2)	**128**
SSIM Parameter Calc	**11**
SSIM Calc	**14**
Map Calc	**1**
MEM Write Map	**1**
Total FLOP / Max MEM	**540 / 2**

To address the performance issues, the ATCS solution can be improved with a multi-phase reduction calculation where each thread loads into shared memory the computation for each SSIM parameter and calculates the sum of each one at a time. The proposed solution would reduce the number of operations performed and reduce the amount of shared memory needed. An additional optimization that would possibly improve performance is increasing the SSIM window to 16x16. Further investigation is necessary to determine the change in quality but the MM solution showed that the higher thread count will be beneficial.

VIII. Gaussian Blur Optimization

A. Gaussian Blur Kernels

The main optimization strategies used for the Gaussian Blur algorithm were:

- Separable Convolution
- Shared Memory
- Constant Memory

Using these strategies we have implemented two versions of the algorithm:

- Basic Solution
  - This version is a simple conversion of the serial code. Each thread generates the Gaussian Blur kernel needed to convolve the output pixel.
- Separable Convolution
  - Splits the 2D Gaussian Blur Convolution into two 1D convolutions that reduce the number of operations performed per output thread from 7x7 to 7+7. Shared memory and Constant memory is also utilized in this version, providing significant memory access reduction.

B. *Gaussian Blur Results*

The Separable Convolution kernel performed up to 65x faster than the basic implementation. Table VII shows the execution time of the 3 kernels with their most optimal block configurations.

TABLE VII
GAUSSIAN BLUR EXECUTION TIME

Kernel (Block Dim)	Total Pixels [MP]	Time [ms]
Basic (16x4)	1.2288	4.45
Horizontal (256x1)		0.0465
Vertical (8x32)		0.0491
Basic (8x16)	11.0592	50.08
Horizontal (256x1)		0.3746
Vertical (8x32)		0.3962
Basic (8x4)	30.72	65.58
Horizontal (256x1)		1.033
Vertical (8x32)		1.087

C. *Gaussian Blur Analysis*

The Separable kernels both utilized shared memory, filling in a tile of input values along with additional halo values needed to convolve the output values. This shared memory tile configuration is based on the block configuration used to run the kernel.

For the Horizontal kernel, a block configuration of 256x1 performed the best across multiple input sizes. This configuration would require a shared memory block of (256+7-1)x1. This maximizes the number of coalesced memory accesses performed by the block when filling in the shared memory tile. It also ensures that the tile size is significantly larger than the convolution mask which provides significant memory access reduction.

For the Vertical kernel, the block configuration of 8x32 performed the best across multiple input sizes. This configuration would require a shared memory block of 8x(32+7-1), which strikes the right balance of partial coalesced memory accesses (8 sequential reads) with a large enough tile size relative to the mask size which also provides significant memory access reduction.

Implementing the Gaussian Blur as a Separable convolution not only reduces the number of operations performed per output thread, it also reduces the number of memory accesses required per output thread. The Basic implementation required 7x7 global memory accesses per output thread, where as a Separable implementation would require 7+7 global memory accesses. Implementing the Separable kernel along with shared memory reduces the total number of memory accesses to [(256+7-1) + 8x(32+7-1)]/256 = 2.210. Meaning we have achieved an overall memory reduction of 49/2.210 = 22.16 when compared to the naive implementation.

When looking at GPU utilization, both the Horizontal and Vertical convolution kernels are memory bound as each thread does not perform a significant number of FLOPS, 15 each. The Horizontal and Vertical kernels achieve a maximum of 1.357 TLFOPS and 1.255 TFLOPS respectively. This is 6.93 and 7.60 times less than the maximum number of FLOPS the Tesla P100 GPU can perform.

IX. OPTIMIZED SOLUTION

The optimized solution comprised the kernels that performed the best. For the optimized solution two FPS were measured, the total FPS and the compute FPS. The total FPS was the time needed to upscale the input frame including memory transfers. The compute FPS on the other hand only includes the time after memory transfers which focuses solely on the time of the kernels.

A. *Optimized Results*

As seen in Table VIII, the optimized solution displayed significant improvements over the Naive solution and Serial solutions. The optimized solution generated 4 to 7 times more frames than the naive solution and 400 to 1600 more frames than the serial solution. The frame rates generated by the optimized solution are in a usable range as some of the upscaled resolutions produce effective frame rates of 60 FPS or more depending on the upscaling factor, total FPS > 120.

TABLE VIII
OPTIMIZED SOLUTION FPS

Input Image	Scale Factor	Optimized FPS		Speedup	
		Compute	Total	Naive	Serial
640x480	2	1412.9	395.2	4.4	442.4
	4	365.3	174.1	5.6	780.8
	6	157.8	83.4	5.7	847.6
1280x720	2	487.0	150.2	4.3	505.8
	4	122.9	62.5	6.3	844.6
	8	52.8	29.3	6.5	1585.9
1920x1080	2	213.4	79.7	4.9	603.1
	4	54.4	27.3	6.3	829.5
	8	23.4	13.2	6.7	1607.8

B. *Optimized Analysis*

The greatest disparity in execution time is the compute and total FPS. If we only consider the upscaling kernels, the FPS is significantly better, 2 to 3 times more. Due to this, we investigated how much time each kernel and memory operation took in the execution time, Fig. 7.

The largest percentage of execution time was dedicated to memory transfers between the host and device. Furthermore, if we view the execution timeline, Fig.8, all the kernels are executed serially with the memory transfer occurring at the

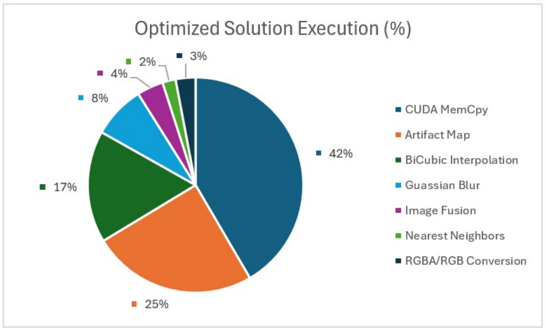

Fig. 7. Percentage of Execution Time in the Optimized Solution

end. With such a long memory transfer time, multiple kernels could execute during that time while the GPU transfers data back to the host, a problem streams can address.

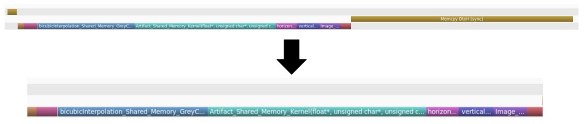

Fig. 8. Execution Timeline for Optimized Solution

X. Streamed Solution

The primary drawback of the optimized solution was the difference between the frame rate realized and theoretical frame rate if no memory transfers occurred. To address this, multiple execution streams can interleave the execution of kernels from different streams while one undergoes a memory transfer, increasing the throughput of the system overall.

A. Stream Experiment

When adding streams the ideal number of streams was determined experimentally by measuring the frame rate achieved for various stream values. The best stream count was selected by choosing the first stream that achieved the maximum frame rate. Fig. 9 showcases the execution rate for a 1280x720p image at various scaling factors from 1 to 6 streams.

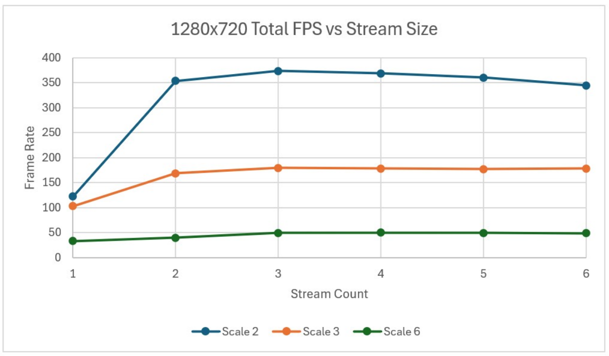

Fig. 9. FPS for various stream sizes with an input picture of 1280x720

From Fig. 9, three streams proved to be the best as higher stream counts either reduced performance as seen in the 2x scale or stagnated as in the other scale factors.

B. Streaming Solution Results

The streamed solution was designed to improve the throughput of the program rather than increasing the speed of the computation of each frame. Figure 10 presents multiple kernels that execute on the GPU while the GPU undergoes a memory transfer.

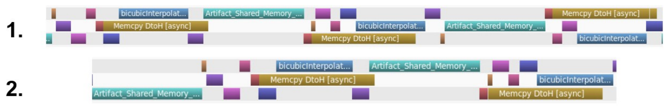

Fig. 10. Streaming Timeline

Table IX contains the FPS for the three-stream solution and the speedup versus the original optimized solution. The compute FPS demonstrated on average no speed up when compared to the non-streamed solution while the total FPS had a speed up of 1.6 to 2.5.

TABLE IX
STREAMING SOLUTION FPS

Input Image	Scale Factor	Streamed FPS		One Stream Speedup	
		Compute	Total	Compute	Total
640x480	2	982.4	638.5	0.70	1.6
	4	376.6	302.2	1.03	1.7
	6	161.8	144.3	1.03	1.7
1280x720	2	493.0	373.6	1.01	2.5
	4	124.3	112.7	1.01	1.8
	8	29.7	28.0	1.01	1.7
1920x1080	2	211.9	181.6	0.99	2.3
	4	55.2	51.6	1.02	1.9
	8	13.2	12.6	1.01	1.7

C. Streaming Solution Analysis

From our results, the streamed solution did not improve the computational FPS but improved the total time for the program. This was expected because both solutions utilize the same kernels but the streamed solution has better throughput (fig. 9). Interestingly, the scale factor of 2 for the smallest image showed a slowdown for its compute FPS whereas other scale factors and input sets averaged no speedup. In our experimentation, this kernel displayed erratic behavior and is likely due to having less time to overlap kernels due to transferring less data overall. Regardless, the streamed solution enabled the entire program to execute faster and produce a higher perceived frame rate.

XI. Conclusion

Table X contains the execution time the serial and GPU enabled solutions for the image fusion algorithm. CUDA and various optimization strategies enabled speed-ups of 700x to 1500x times that of the serial solution and frame rates ranging from 100 FPS to 600 FPS for common resolutions. When compared to the Naive solution, the optimizations employed achieved an additional 6-12x speed up.

TABLE X
FPS COMPARISON FOR ALL SOLUTIONS

Input Image	Scale Factor	CUDA			Serial Code	Speedup		
		Streamed	Optimized	Naive		vs Streamed	vs Naive	vs Serial
640x480	2	638.5	395.2	89.6	0.893	1.6	7.1	715.0
	4	302.2	174.1	30.9	0.223	1.7	9.8	1355.3
	6	144.3	83.4	14.5	0.098	1.7	9.9	1472.4
1280x720	2	373.6	150.2	35.1	0.297	2.5	10.6	1257.8
	4	112.7	62.5	9.9	0.074	1.8	11.4	1522.9
	6	28.0	16.9	4.5	0.018	1.7	6.3	1558.2
1920x1080	2	181.6	79.7	16.2	0.132	2.3	11.2	1375.4
	4	51.6	27.3	4.3	0.033	1.9	11.9	1563.9
	6	12.6	7.6	2.0	0.008	1.7	6.4	1573.8

A. Applications

As mentioned at the start of the paper, we need to consider both the native rendering FPS of the game and in addition to the upscaling FPS. If a 640x480p game is rendered at 120 FPS and upscaled by a factor of 4, the upscaling FPS would be approximately 638 FPS. Using equation (1), the effective FPS is found to be:

$$\text{Effective FPS} = \frac{638 * 120}{638 + 120} \approx 101 \text{ FPS}$$

The fusion based algorithm enables video game developers to improve the performance of their games while increasing the resolution of their images. Fig. 11 showcases an example of a 2x upscaled frame from the video game ”Luigi’s Mansion” (2001).

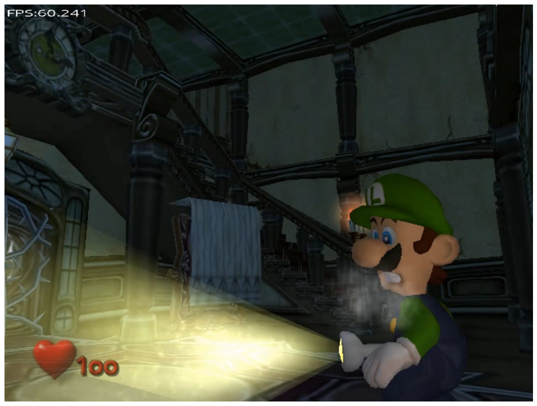

Fig. 11. Luigi’s Mansion (2001) Upscaling in Real Time

B. Future Work

One improvement to the algorithm to investigate is a replacement algorithm for nearest neighbors. A more powerful upscaling algorithm could replace it and produce higher quality images and leave more opportunities to optimize it. Nearest neighbors is lightweight both at a benefit and detriment. The algorithm should be merged with the BiCubic kernel to reduce redundant memory accesses because the same shared memory space could be used to calculate both images.

The two kernels that comprised the largest percentage of execution time were artifact map generation and BiCubic interpolation and are worth spending the majority of time optimizing. The artifact map’s SSIM calculation contains many redundant computations that can be improved by using a multi-phase reduction algorithm to calculate each of the SSIM sums. A solution like this will reduce the FLOP count per thread and improve the FLOP to bandwidth ratio. BiCubic interpolation could be implemented as a separable convolution similar to the Gaussian blur which can reduce the number of FLOP needed to perform the upscale.

Finally, 2x upscaling has interesting behavior with certain kernels performing differently than the proposed ”one size fits all” solution. If the application is targeting a specific resolution needing a 2x scale, the solution can be modified to use specific versions of our kernels or use the 2x upscaling configuration twice to upscale an image 4x rather than one 4x operation.

C. Source Code

The source code along with test benches and instructions for creating datasets can be found on the author’s GitHub repository.[1]

ACKNOWLEDGMENT

Special thanks to Ali Akoglu for advice and suggestions for the project and paper.

REFERENCES

[1] B. Zhang, T. Xu, Y. Chang, Z. Li and Y. Li, ”Design and Implementation of Image Signal Processor Based on CUDA,” 2022 7th International Conference on Signal and Image Processing (ICSIP), Suzhou, China, 2022, pp. 222-227, doi: 10.1109/ICSIP55141.2022.9887249.
[2] X. Liu et al., ”Efficient Bicubic Interpolation Architecture for RGB Image Data Stream,” 2023 6th International Conference on Electronics Technology (ICET), Chengdu, China, 2023, pp. 1356-1361, doi: 10.1109/ICET58434.2023.10211590.
[3] X. Kang, S. Li and J. Hu, ”Fusing soft-decision-adaptive and bicubic methods for image interpolation,” Proceedings of the 21st International Conference on Pattern Recognition (ICPR2012), Tsukuba, Japan, 2012, pp. 1043-1046.
[4] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli, ”Image quality assessment: from error visibility to structural similarity,” in IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.
[5] “Nvidia DLSS 4 Technology,” NVIDIA, https://www.nvidia.com/en-us/geforce/technologies/dlss/ (accessed Feb. 25, 2025).
[6] M. Fan, X. Zuo and B. Zhou, ”Parallel Computing Method of Commonly Used Interpolation Algorithms for Remote Sensing Images,” in IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing, vol. 17, pp. 315-322, 2024, doi: 10.1109/JSTARS.2023.3329018.

[1]https://github.com/Matthewt1441/Fusion_Image_Upscale_HPC

[7] H. Li, D. Yu, A. Kumar and Y. -C. Tu, ”Performance modeling in CUDA streams — A means for high-throughput data processing,” 2014 IEEE International Conference on Big Data (Big Data), Washington, DC, USA, 2014, pp. 301-310, doi: 10.1109/BigData.2014.7004245.

[8] B. Zhang, T. Xu, Y. Chang, Z. Li and Y. Li, ”Design and Implementation of Image Signal Processor Based on CUDA,” 2022 7th International Conference on Signal and Image Processing (ICSIP), Suzhou, China, 2022, pp. 222-227, doi: 10.1109/ICSIP55141.2022.9887249.

[9] F. Jiang, D. Shi and D. C. Liu, ”Fast Adaptive Ultrasound Speckle Reduction with Bilateral Filter on CUDA,” 2011 5th International Conference on Bioinformatics and Biomedical Engineering, Wuhan, China, 2011, pp. 1-4, doi: 10.1109/icbbe.2011.5780213.

[10] G. Lan, Y. Shen, T. Chen, and H. Zhu, “Parallel implementations of structural similarity based no-reference image quality assessment,” Advances in Engineering Software, vol. 114, pp. 372–379, Aug. 2017, doi: https://doi.org/10.1016/j.advengsoft.2017.08.003.

[11] L. M. Russo, E. C. Pedrino, E. Kato and V. O. Roda, ”Image convolution processing: A GPU versus FPGA comparison,” 2012 VIII Southern Conference on Programmable Logic, Bento Gonçalves, Brazil, 2012, pp. 1-6, doi: 10.1109/SPL.2012.6211783.

[12] Victor Podlozhnyuk, Image Convolution with CUDA, nvidia CUDA 2.0 SDK convolutionSpeparable document

[13] Nintendo EAD. (2001). Luigi’s Mansion [Nintendo GameCube]. Nintendo

[14] Wikipedia: The Free Encyclopedia. Accessed May 3, 2025. [Photo]. Available: https://en.wikipedia.org/wiki/Bicubic_interpolation